SoloRack SDK v0.11 Beta Documentation

Dependencies and prerequisites

1. A C++ compiler. (Preferably VC++, I haven't tested other compilers yet).
2. VSTGUI version 3.6. (Already included). Also available here:

https://sourceforge.net/projects/vstgui/files/vstgui/VSTGUI%203.6/

Note: VSTGUI is released under a completely different license than VST™. VSTGUI is open source and doesn't require a signed agreement. The license can be found in its source code files.

Creating your first module

The whole SDK code is wrapped in a single Visual C++ project with few test modules that you can edit to create your own modules. If you don't have VC++, there is a free version called "Visual Studio Community edition". Or if you're like me who likes old and CPU light software, "Visual Studio 2008 Express" is an option. There is still an official download link for it floating around. The following applies to VC++ users.

1. First make sure you have the latest version SoloRack.

2. Obviously extract the SDK. A folder named "SoloRack SDK v0.11" will be created.

3. In the "vstgui patch" folder. You'll find 4 files that are my modified versions of the vstgui.h, vstgui.cpp, vstcontrols.h and vstcontrols.cpp. Copy these to the "vstgui" folder to replace the existing ones. (You may want to backup the originals).

I tried to keep my changes to vstgui minimal. If you want to know what changes I did, just search for "ammar" inside any of these 4 files and you'll find some comments that indicate each and every change I made.

Note here these 4 files mentioned above are not considered part of the SoloRack SDK. These are modified from VSTGUI and so, still are distributed under the VSTGUI license which you can read clearly inside those files. I DO NOT claim to be the author of these files.

4. Copy the **TestModules** folder to the **Modules** folder where SoloRack.dll exists. In case you are using SoloRack 64bit, then copy **TestModules.x64** to the **Modules.x64** folder where SoloRack.x64.dll exists.

5. Copy the **Skin** to the **TestModules** (or **TestModules.x64**) folder you just copied.

6. Open the project/solution file **TestModules.sln** in VC++. It will probably ask you to convert the project to your newer version (since I use an old version, Visual Studio 2008).

7. Go to [project properties] > [General] > [Windows SDK version], make sure you select the latest version listed (like 10.x) instead of 8.1.

8. Make sure you choose the release platform you want (i.e. x64 or Win32 (x86)). Then Compile/Build it. This should produce the **TestModules.dll (or TestModules.x64.dll**) which contains all of the test module(s).

9. After a successful compilation. Copy the **TestModules.dll** file to the **Modules** folder where SoloRack.dll exists. In case you are using SoloRack 64bit, then the dll is **TestModules.x64.dll.** It should be copied to the **Modules.x64** folder where SoloRack.x64.dll exists

10. Start SoloRack and you should see the example test modules placed in the modules menu (Currently there is a mixer and tempo from DAW module).

The **TestModules.ini** contains basic information about the modules contained in the **TestModules.dll** file. SoloRack reads these .ini files when it starts instead of scanning/loading each dll in the modules folder which can cause crashes and delays in load time. Developers are responsible of providing an .ini for each dll they distribute. The ini file should have the same file name as the dll but obviously with a .ini extension instead of a .dll.

Also a user may edit an ini file to hide some modules that he/she doesn't want to use or change the order in which they appear in the menu.

Classes provided

SoloRack inherits most of its C++ classes from VSTGUI. The **Module** class inherits from **CViewContainer** and adds audio processing functions to it that will be called by the audio thread. Other classes include, **PatchPoint, ModuleKnob, ModuleKnobEx, CKickButtonEx, CMovieBitmapEx, CSpecialDigitEx** and **Product**.

Programming interface

Static functions of your module(s) are reached by SoloRack using the pointers stored in the **DllModule** struct. *Virtual* functions of your module are reached using a wrapper class that forces the correct vtables to be inherited.

On the other hand, your modules can talk to SoloRack using the function pointers stored in the **SynthComm** struct (use the synth_comm variable).

The following functions need to be implemented by your Dll modules. These are called by SoloRack (Except for **Initialize()** and **End()** which should be called by your Dll). If you don't implement some of the functions, there is usually a default behaviur function that is called automatically. These defaults are defined in the parent **Module** class.

- **Initialize()**

This is the first function that should be called for each module after your Dll is loaded. Common or global preparation and initialization should be performed here, like for example allocating bitmaps and memory that will be used by all instances of a module.

Calling this function is the responsibility of your Dll. Typically call it inside DllInitialize() which SoloRack calls after DllMain() is called at the process attach.

- **End()**

This is the last function called before the Dll is unloaded from SoloRack memory. Destruction and freeing of common or global memory should be done here.

Calling this function is the responsibility of your Dll. Typically call it inside DllMain() at the process detach.

Mandatory functions

The following is a list of the most important functions that MUST be implemented by your modules. These are called by SoloRack when needed.

- **Constructor()**

Because there is no direct way in C++ to get a pointer to a class constructor. This function should be implemented to create a new instance of a module and return it back to SoloRack.

- **GetName()**

Should return a string (char *) that indicates the full name of the module. This name combined with your vendor name constitutes a unique identifier for your module.

- **GetNameLen()**

As the name suggests

- **GetVendorName()**

Should return a string that indicates your vendor (company) name. This name

combined with the module name constitutes a unique identifier for the module.

- **GetVendorNameLen()**

As the name suggests

- **GetVersion()**

Should return the version number of the module. This number is automatically saved with presets. And given back to the module when opening a preset using LoadPreset(). This will allow you to achieve backward compatibility with presets that has been saved with an older version of your modules.

- **GetType()**

This function is called by SoloRack and must be implemented in your module. It Returns the module type. Used to categorize it in the module menu. The following types are provided:

```
kMixerPanner
kFilter,
kOscillatorSource,
kAmplifier,
kModulator,
kModifierEffect,
kClockGate,
kSwitch,
kCVSourceProcessor,
kFromToDAW,
kSequencer,
kLogicBit,
kMIDI,
kOther
```

These types are also used in the .ini file but without the “k” in-front of each type. For example, say "VCF1" and "VC SEM" are two filters and "Rad Reverb" is an effect. The Ini file would look something like this:

```
[Configuration]
Vendor = Your Company or Author Name

[Filter]
number_of_modules=2
M1 = VCF1
M2 = VC SEM

[ModifierEffect]
number_of_modules=1
M1 = Rad Reverb
```

- **Activate()**

Tries to activate the module given license information parameters. You can just return NULL if this is not relevant.

- **IsActive()**

Checks and returns true only if the module is active (licensed). Otherwise, returns false.

- **GetProductName()**

Should return your own product name for the module. Product names should include the vendor name to avoid conflicts with other vendor modules in the config.ini

Optional Functions

The following is a list of most of the *virtual* functions that can be implemented by your modules. These are called by SoloRack when needed. However, you may choose to leave some of them unimplemented, as thier functionality may not be required by your module(s), In this case, the parent (Module class) version will be called. Which in many cases, just does nothing.

- **ProcessSample()**

This is the one and only audio processing function. It should take values from the input patch points. Process them, and put the output values to the output patch points. This is done sample by sample. That is, each time ProccessSample() is called, it should process only one value (sample) which exist at the inputs.

This function is time critical as it is called at audio rate. It's essential that you try to optimize it as much as possible. Lengthy pre-calculations should be left outside if possible. Function calls should be minimal. Inlining may speed up things, but note that your compiler may not listen to your **inline** keywords. Infact, VC++ doesn't inline most of the time (as far as my tests go). I some times prefer preprocessor macros, as they are guaranteed to stay inline.

This function automatically takes ownership of the audio critical section before it's called to assure mutual exclusion when two events happen that need to access the same data. For example, when a cable is disconnected. This function will not be called until the disconnection is finished and all related modules informed.

- **ProcessEvents()**

Tihs is called whenever the DAW sends an event (like a MIDI note, etc) to SoloRack. Note that your module has to call **CallProcessEvents()** once to inform SoloRack that it needs to process DAW events. Otherwise, SoloRack will NOT call this function for your module. This is done to minimize CPU, as most modules will not need to process DAW events. You typically call CallProcessEvents() in the constructor.

This function automatically takes ownership of the audio critical section before it's called. In other words, this function will not be called while ProccessSample() is executing. And ProccessSample() will not be called while this function is executing. i.e, they are mutually exclusive.

- **StartOfBlock()**

Every time the DAW calls processReplacing() in SoloRack (which processes a block of samples). SoloRack will call StartOfBlock() once for every module that requires this functionality. To indicate this, your module has to call CallStartOfBlock() **once**. Typically called in your constructor.

This function gives a chance for your module to do work that it may not need to do so often (i.e not at audio rate). Like for example calling, the SynthComm GetDAWTime() function to get sync and tempo information. Or doing heavy calculations that doesn't have to be done at the audio sampling rate.

This function will be called before the first ProccessSample() is called at the start of the block.

- **CableConnected()**

Whenever a new cable is connected to one of a module's patch points. This function will be called. This gives your module a chance to precalculate and do necessary changes. Note that this call takes ownership of the audio processing critical section. So it's safe to assume that ProcessSample() is not running while CableConnected() is being executed.

- **CableDisconnected()**

Whenever a cable is disconnected from a module's patch point. This function will be called. This gives your module a chance to precalculate and do necessary changes. Note that this call takes ownership of the audio processing critical section. So it's safe to assume that ProcessSample() is not running while CableConnected() is being executed.

- **SetSampleRate()**

This is called whenever SoloRack's internal sampling rate changes. It changes when either the DAW sampling rate changes or SoloRack's oversampling settings changes. This gives the module a chance to recalculate relevant information.

Note that if your module has band limiting features (like a VCO). This function should call SetBandLimit() to accommodate for the change if required, or at least do necessary calculations inside SetSampleRate itself.

This function takes ownership of the audio processing critical section. So it's safe to assume that ProcessSample() is not running while this function is being executed.

- **SetDAWSampleRate()**

This is almost exactly like SetSampleRate() with the difference that it's only called when the sample rate change happens in the DAW. And Its always called before SetSampleRate() is called.

- **SetDAWBlockSize()**

Called wherever DAW block size changes.

- **ValueChanged()**

This is called whenever a control (knob, switch, etc..) value has changed either by the user or by DAW automation. This is the only function where ProcessSample() may be running WHILE this function is running. To ensure that this doesn't happen, you have to manually take ownership of the audio processing critical section by calling EnterProcessingCriticalSection() inside this function. You have to also call LeaveProcessingCriticalSection() when you are finished, otherwise audio processing will hang!!.

I could have done this automatically. But I finally decided to leave it to the developer to decide for performance reasons. Because DAW automation may cause multiple frequent calls to this function which will cause ownership of the critical section to be taken so frequently that it may affect audio processing and lead to buffer underuns in the worst case.

- **SavePreset()**

This is called when a preset is to be saved, giving a chance for the module to do it's own none-default saving. The function is given an empty buffer (chunk) allocated by SoloRack to save it's data. SoloRack will know the required buffer size by calling GetPresetSize() in your module which should return the size required.

When this function is not implemented, the default SavePreset() in the Module class calls SaveControlsValues() which automatically saves all control **values** (knob, switch, etc.. values) in the module.

If you want to change this default behavior, you should implement this function. Ofcourse you can make calls to SaveControlsValues() inside your own SavePreset() and add your own special data to the buffer.

Note: the order in which SaveControlsValues() saves the control values is related to the order in which the VSTGUI addView() function was called in your constructor. This is important, because in latter versions of your Module, if you

change that order, then loading old presets would not work correctly, you have to correct that by checking the version number of the preset. This is possible. But its offcourse prefered that you don't change that order in latter versions so that backward compatibility becomes easy for you.

- **LoadPreset()**

This is called when a preset is to be loaded, giving a chance for the module to do it's own none-default loading. The function is given a data buffer (chunk) allocated by SoloRack which contains the preset data to be loaded. The size of the data is given too. Also the version number of the module that originally saved the preset will be given. You can check the version number to achieve proper backward compatibility.

When this function is not implemented, the default LoadPreset() in the Module class calls LoadControlsValues() which automatically loads all control **values** (knob, switch, etc.. values) in the module.

If you want to change this default behavior, you should implement this function. Ofcourse you can make calls to LoadControlsValues() inside your own LoadPreset() and still load your own special data from the data buffer.

Note: the order in which LoadControlsValues() saves the control values is related to the order in which the VSTGUI addView() function was called in your constructor. This is important, because in latter versions of your Module, if you change that order, then loading old preset would not work correctly, you have to correct that by checking the version number of the preset. This is possible. But its off-course preferred that you don't change that order in latter versions so that backward compatibility becomes easy for you.

- **GetPresetSize()**

This is called just before **SavePreset()** is called. Your module should return the exact size (in bytes) of how much memory should SoloRack reserve for your module's preset/data.

- **GetInfoURL()**

This should return a string containing a URL that points to a web page that provides more information/documentation for the user about your module.

- **GetAlwaysONDefault()**

By default, to save CPU, SoloRack stops audio processing a module if it's no longer connected by any cable. The user can change that behavior by right clicking and choosing "Always ON". This function provides the default value for this option.

In other-words, this function should return true only if the module (by default)

requires ProcessSample() to be called all the time regardless of whether or not the module is connected. This is useful for example, for modules that have visual feedback that is constantly changing like LFO LEDs, scopes, or modules that are time dependant.

- **IsAudioToDAW()**

Should return true if the module can send Audio/CV to the DAW, otherwise it should return false. SoloRack needs to know this beforehand for it's internal preparation.

- **InstanceActivate()**

Is just an instance version of the static function Activate() mentioned above. This is to make things easier when activating by right click.

- **InstanceIsActive()**

Is just an instance version of the static function IsActive() mentioned above.

- **AddDemoViews()**

This is part of the activation system. It gives a chance for the module to make any internal preparation including any visuals that need to be displayed when demo mode is enabled.

- **SetEnableDemo()**

This is part of the activation system. It is called whenever demo mode needs to be enabled or disabled.

- **SetBandLimit()**

This is called when the user changes the "band limit" choice in the module right-click menu. This is meant mainly for VCOs and sound sources that provide internal anti-aliasing features. Your module should decide what to do with it. The two options currently available are "At oversampling nyquist" and "Near DAW nyquist".

- **PolyphonyChanged()**

This is called whenever the user changes the number of voices in the main menu

- **onMouseDown()**
- **onMouseUp()**
- **onMouseMoved()**

These come from VSTGUI. The default implementation calls SoloRack internal functions through the synth_comm variable. You can add you custome handling here. But you better also keep calling SoloRack's own functions because otherwise your modules won't respond to mouse correcrtly.

- **OnMouseMovedObserve()**

This one is NOT from VSTGUI. It allows your module to monitor (observe) mouse

movements even when the mouse is not hovering on top of the module. To let SoloRack call this, you have to call CallMouseObserve() once.

- **GetIsMonoDefault()**

Should return false only if you want your module to be a polyphonic module by default.

- **ConstructionComplete()**

This function is called by SoloRack after all constructors of all voices of a module is called. And after all solorack specific information like index, procindex, etc... has been setup. This is where GetVoiceModule() will start working. The purpose of this function is to give your module a chance to do initialization work that is global for all voices.

- **DestructionStarted()**

This function is called by SoloRack for all voices of a module just before destructors are called for all voices of that module. This will give you a chance to do clean up work that is global for all voices.

This also gives a chance for odd/special type of modules that cannot imidiatly destruct, for example in case the module is running a high CPU worker thread that needs to be stopped, or is connecting to the network. The idea here is to give all modules a quick heads up, then atempt to destruct them one by one. This allows those odd/special modules to end their lengthy work WHILE solorack destructs other modules.

- **IsPolyManager()**

A poly manager module is one that is responsible for chaining polyphonic events to all voices in a polyphonic patch. This chaining is done either through MIDI or CV patch points. The **pnext** variable has to be used for this purpose. Each patch point points to the next patch point in the chain. The “SD05 MIDI Poly Chainer” is a poly manager.

This function should obviously return **true** if your module is a poly manager.

Helper Functions

The following list of functions are already implemented and are meant to make your life a bit easier. These are NOT called by SoloRack. They are for your own use.

- **PutLeftScrews()**
- **PutRightScrews()**

As the name suggest. Just try them.

- **InitPatchPoints()**

Goes through all the patch points you added to your module and sets the values of **In** and **out** to given value.

- **EnterProcessingCriticalSection()**
- **LeaveProcessingCriticalSection()**

These two function are used to deliberately enter and leave SoloRack's audio critical section. While your thread is inside the critical section (ie. taking owner ship of it). ProccessSample() is guaranteed to NOT be called.

You only need those two functions inside ValueChanged() or inside onMouse..() functions. You don't need them for any other functions like CableConnected() or CableDisconnected(), Destructors, ProccessEvents(), StartOfBlock() etc.. since all these are all guaranteed to not be called while ProccessSample() is executing.

Obviously, you don't need to call these two functions either if your not accessing data that is shared between the audio and GUI thread. Also remember that most primitive types are atomic on many systems. So, sometimes you can getaway even if you are accessing shared data.

- **UpdateSValue()**

This is part of ModuleKnob class. It's meant to be use for parameter smoothing. It will take **value** variable of the knob and smooth it's changes and put the output in **svalue** variable.

- **UpdateSValueReached()**

Same as UpdateSValue() but returns a boolean. True only when **svalue** becomes equal to **value.**

- **ModuleKnobExPool class**

This class is solely meant for CPU performance. When you have too many knobs in your module that require smoothing. It is more CPU efficient to use this pool class than the UpdateSValue() functions. Read the comments in the code on how to use it.

- **GetCurrentStep()**

This is part of ModuleKnob class. Used for stepping knobs, not continuous knobs. i.e. If the knob has the **is_stepping** variable equal to true. This function can be used to return the step number. Which ranges from zero to **subPixmaps**-1.

- **SendAudioToDAW()**

This is actually a set of overloaded functions. There are 5 of them. The main reason for having so many is mainly CPU usage efficiency.

- **ReceiveAudioFromDAW()**

Another set of overloaded functions for getting audio/CV from the DAW.

- **SetKnobsSmoothDelay()**

You can use this to set the how long the parameter/knob smoothing takes.

- **GetVoiceModule()**

When your module is set to polyphonic mode, this function returns a pointer to a specific voice module that you request. This allows you for example, to control all voices from one voice zero. Or do customized behavior per voice.

- **AddPatchPoint()**
- **AddMIDIPatchPoint()**
- **AddModuleKnob()**
- **AddVerticalSwitch()**
- **AddMovieBitmap()**
- **AddKickButton()**
- **AddOnOffButton()**
- **AddSpecialDigit()**
- **AddSpecialDigitEx()**
- **AddModuleKnobEx()**
- **AddTextLabel()**
- **AddHorizontalSlider()**

This collection of functions are to simplify adding controls to your modules. They automatically implement DAW automation. You can however if you want, use VSTGUI directly and add your own controls or customized controls.

- **GetFreeTag()**
- **RegisterTag()**
- **UnRegisterTag()**

These three functions are meant to support DAW automation of your parameters/controls. You don't need them most of the time if you use the earlier mentioned Add…() functions.

- **SetForceMono()**

Used to force a patch point to be mono regardless if the module was mono or not!!

- **ClearIfOrphaned()**

Should only be called in the destructor or just before it.

SoloRack calls forget() when the user wants to delete modules. forget() will destruct and delete the module if **nbReference==0**. That is, nothing else left thats pointing to that module. But in very odd cases, you may not want to imidiatly destruct your module after it's been deleted. For example, with modules that connect to the network or have worker threads activly working. This can be done calling remmember() early, then calling forget() when work has finished and you are ready to destruct. However, during this "hanging there period" SoloRack will realise that the module didn't destruct because forget() didn't return true. So SoloRack will put the module into an orphaned list that will be forced to destruct at SoloRack exit.

This ClearIfOrphaned() function tells SoloRack that this module has gracefully been detructed by you. So if it's in the orphaned list. SoloRack will remove it and not try to destruct it on synth exit.

You HAVE to call this function if you automatically destruct your modules later (using forget()). Otherwise SoloRack WILL CRASH on synth exit. Because it will try to decontruct your modules which doesn't exist any more in memory.

- **GetDAWBlockSize()**

Returns the current DAW block size in samples.

- **GetDAWSampleRate()**

Returns the current DAW sample rate.

SynthComm Functions

The SnythComm structure contains pointers to internal SoloRack functions that give you a way to communicate with the synth. The *synth_comm* variable is already initialized for that purpose and can be used directly by your modules.

- **GetSynthSDKVersion()**

Get the SDK version that SoloRack was built on.

- **GetEditor()**

This will return a pointer to SoloRack's editor. This function is already called for you in Module constructor. The editor pointer is put in the variable *peditor*. You'll need this pointer if you want call some of the following functions.

Some of those functions have the same name and functionality as some helper functions mentioned above. This might seam like an unnessesary redunduncy. But It's done this way on purpose.

- **GetSynth()**

Returns a pointer to the SoloRack synth itself. This is not usefull unless you have a VST2 license and therefore can treat the returned value as a VST2 synth. You

will not need it any way as the other functions already cover many needs.

- **GetOversamplingFactor()**

The oversampling factor is the ratio *sample_rate / DAW_sample_rate*. The user chooses this.

- **GetVoiceModule()**

This is similar to the helper function mentioned above. Infact the helper is a short cut to this.

- **GetPolyphony()**

Returns the current number of voices.

- **ClearIfOrphaned()**

This is the same as the helper function mentioned above. Infact the helper is a short cut to this.

- **GetDAWBlockSize()**

Returns the current DAW block size in samples directly from SoloRack. This value is already stored in a Module variable that you can read using the helper function named with the same name GetDAWBlockSize()

- **GetDAWSampleRate()**

Returns the current DAW sample rate directly from SoloRack. This value is already stored in a Module variable that you can read using the helper function named with the same name GetDAWSampleRate()

- **GetDAWTime()**

Provides a way to get information about current DAW timing. Like **ppq** position. Tempo, time signature etc..

Events Handling

The following few functions are meant to be used inside ProcessEvents(). They are mostly self explenatory when you look at the SDK code.

- **GetNumberOfEvents()**
- **GetEventsArray()**
- **GetEventType()**
- **GetEventTime()**
- **GetMIDIEventMessage()**
- **GetMIDISysExSize()**
- **GetMIDISysExMessage()**

Other SynthComm functions

There are many other functions that relate to mouse and patch point handling.

Many of those are used by some helper functions or other GUI functionalities. You don't need to explicitly use them most of the time.

A word on ABI compatibility

The ABI (Application Binary Interface) is very dependant on the compiler. Since we can't control possible future compiler changes, this means that we can't guarantee ABI compatibility between SoloRack and your Dll modules if both don't use the same compiler version. In other words, a deep compiler change could possibly break ABI compatibility. Furthermore, if we're not careful, changes to the SDK API itself can break the ABI compatibility. Well that's, in theory.

Having said that, that doesn't mean we can't do something about this that would minimize the possibility of that happening. Otherwise, technologies like COM won't work. The SoloRack SDK uses the following methodology:

1. Newer functions/variables added to the API in later versions are only added to the **END** (i.e tail) of Module and ModuleWrapper classes. This is crucial because, (for historical reasons), it seams that most if not all compilers keep the order of the binary interface as it is mentioned in the C/C++ definitions/declaration. So when we add to the END, we are almost sure that all past code doesn't have to change.

2. When SoloRack launches your module. It calls the C interface exported function GetDllModule() .This should return the SDK version number (among other things) that was used to build your module. If it's a version lower than the one SoloRack is running on, SoloRack will make sure it doesn't call any newer functions that your module's SDK doesn't support. If it's a version higher than the one SoloRack is running on, then the module is NOT launched and the user is informed to download the latest SoloRack.

The above is what WE have to do. Here is what YOU have to do to make sure your Modules don't break the ABI.

1. Never modify the "ModuleWrapper.h" file or the "ModuleWrapper.cpp" file.

2. If you want to add functions or variables to the Module class. The safest way to do so is to derive a new base class from the Module class. Add your functions and variables to it. Then create your own modules deriving from that new base class you created.

3. If for any reason you don't want to make a derived class but you want to add your own functions and variables directly to the Module class. Then you have to add those to the END (i.e tail) of the class declaration. The point here is, ORDER IS RELEVANT. If you change the order in which functions or variables appear. You are most likely going to break ABI compatibility.

*Note: If I'm not mistaken, internally in memory, the END (tail) for **virtual** functions is different than the END for variables. They're stored in different locations. In other words, you can consider them as two separate lists. Even though they are listed in one place in the Module.h file.*

In the worst case, If a future update to the SDK does break the ABI because of major changes. Then the normal remedy would be to just recompile your modules using the newer SDK version.

Now things get more complicated when you use different compilers. Destructors and operators are implemented differently and so can break the ABI too. This is still left for future work. Currently the SDK is only tested and ready for Microsoft VC++ compiler.

Ammar Muqaddas

www.solostuff.net